University of Thessaly

Master's Thesis

Support of PDEs for multidomain problems in a solving environment

Author:
Emmanouil Maroudas

Supervisors:
Christos Antonopoulos
Manolis Vavalis
Nikolaos Bellas

Computer Systems Lab (CSL)
Department of Electrical and Computer Engineering

January 22, 2016

UNIVERSITY OF THESSALY

Abstract

Computer Systems Lab (CSL)
Department of Electrical and Computer Engineering

Master

Support of PDEs for multidomain problems in a solving environment

by Emmanouil Maroudas

Τα τελευταία χρόνια οι τεχνολογικές εξελίξεις στους τομείς του υλικού και του λογισμικού έχουν σηματοδοτήσει την αρχή μιας νέας εποχής για τις εφαρμογές μοντελοποίησης και προσομοίωσης, τόσο στους ακαδημαϊκούς κύκλους, όσο και στη βιομηχανία. Η παρούσα διατριβή παρουσιάζει τη σχεδίαση, υλοποίηση και πειραματική εκτίμηση ενός ενισχυμένου περιβάλλοντος μετα-προγραμματισμού βασισμένου στην πλατφόρμα FEniCS, επικεντρωμένου στην επίλυση προβλημάτων πολλαπλών χωρίων - πολλαπλών φυσικών χαρακτηριστικών (MDMP), τα οποία είναι μοντελοποιημένα με τη χρήση μερικών διαφορικών εξισώσεων (PDEs). Πιο συγκεκριμένα, το προτεινόμενο περιβάλλον βασίζεται σε γλώσσες σεναρίων (Python,) ακολουθώντας μία αρχιτεκτονική προσανατολισμένη σε διαδικτυακές υπηρεσίες. Ο σχεδιασμός καλύπτει ένα μεγάλο εύρος προβλημάτων, ωστόσο η πρότυπη υλοποίηση που παρουσιάζεται, περιορίζεται σε ελλειπτικές PDEs δυο και τριών διαστάσεων. Επιπλέον, γίνεται φανερή η ευκολία ενσωμάτωσης προηγμένων επιλυτών και μεθόδων επίλυσης, όπως αυτοί που προσφέρονται από το περιβάλλον FEniCS [1] και στο deal.II [2, 3], καθώς και η ανάπτυξη νέων εντός του περιβάλλοντος. Ενδεικτικά παραδείγματα αποτελούν οι μέθοδοι χαλάρωσης υποχωρίων με ή χωρίς επικάλυψη και οι υβριδικοί στοχαστικοί/ντετερμινιστικοί επιλυτές [4, 5].

UNIVERSITY OF THESSALY

Abstract

Computer Systems Lab (CSL)
Department of Electrical and Computer Engineering

Master

Support of PDEs for multidomain problems in a solving environment

by Emmanouil Maroudas

Evolution on hardware and software technologies during the last years leads to a new era of scientific modeling and simulation in both industry and academia. This thesis describes the design, implementation and evaluation of an enhanced meta-computing environment based on the FEniCS Project, focusing on multi-domain multi-physics (MDMP) problems modeled with partial differential equations (PDEs). In particular, we propose an enhanced meta-computing environment which is based on: (a) scripting languages (Python) and their practices, and (b) on the Service Oriented Architecture (SOA) paradigm and the associated web services technologies. Although our design is generic, covering a wide range of problems, our proof of concept implementation is restricted to elliptic PDEs in two or three dimensions. Furthermore, it clearly shows that our tool can easily exploit state of the art numerical solvers like those available in FEniCS [1] and deal.II [2, 3], domain decomposition methods with or without overlapping, Monte Carlo based hybrid solvers [4, 5], rectangular or curvilinear domains and interfaces and beyond.

Contents

List of Figures

List of Tables

List of Codes

Abbreviations

DD	**D**omain **D**ecomposition
FEM	**F**inite **E**lement **M**ethod
FEA	**F**inite **E**lement **A**nalysis
PDEs	**P**artial **D**ifferential **E**quation**s**
MDMP	**M**ulti **D**omain **M**ulti **P**hysics
UFL	**U**nified **F**orm **L**anguage

Chapter 1

Introduction

Advances in hardware and software technologies in the 1980s led to the modern era of scientific modeling and simulation. This era seems to come to an end. The simulation needs in both industry and academia mismatch with the existing software platforms and practices, which to a great extent have remained unchanged for the past several decades. We foresee that this mismatch, together with the emerging ICT advances and the cultural changes in scientific approaches will lead to a new generation of modeling and simulation.

This thesis proposes approaches for designing, analyzing, implementing and evaluating new simulation frameworks particularly suited to multi-domain and multi-physics (MDMP) problems that have Partial Differential Equations (PDEs) in their foundations.

These types of problems appear frequently on large scale, complex, real world problems from various science fields. Considering their heavy computational needs, it seems reasonable to facilitate their development, while reducing their execution time using every available device/machine on a system/network.

In particular, we propose an enhanced meta-computing environment which is based on: (a) scripting languages (Python) and their practices, and (b) on the Service Oriented Architecture (SOA) paradigm and the associated web services technologies.

Although our design is generic, covering a wide range of problems, our proof of concept implementation is restricted to elliptic PDEs in two or three dimensions. The implementation clearly shows that our tool can easily exploit state of the art numerical solvers like

those available in FEniCS [1] and deal.II [2, 3], domain decomposition methods with or without overlapping, Monte Carlo based hybrid solvers [4, 5], rectangular or curvilinear domains and interfaces and beyond.

The rest of this thesis is organized as follows: Chapter 2 presents some basic background information about MDMP problems as well as finite element approaches for solving them. Chapter 3 discusses the essential components and features and API of the FEniCS project and justifies the decision to build our platform on top of it. In Chapter 4 we discuss the design decisions, goals and the solving methodologies we aim to support in the platform. Chapters 5 and 6 provide the background of the hybrid Monte Carlo and the Schwarz alternating methods respectively, and their implementation in our platform. We discuss the design and implementation of the prototype web services support in Chapter 7. All the above chapters contain code examples and snapshots from the implementation when necessary, in order for the reader to obtain a complete understanding of our approach. Chapter 8 discusses a use-case, an environmental engineering application setup that utilizes the method described in chapter 6. Chapter 9 concludes the thesis, outlining the benefits of the platform.

Appendix A contains a full example, with the source code, of the Schwarz method for 3D overlapping domains. It also demonstrates the configuration and setup of the problem described in Chapter 8.

Part I

Background

Chapter 2

MDMP problems

Multi-physics problems are encountered when the behavior of a system is affected by the interaction between several distinct physical fields (e.g., structural deformation, fluid flow, electric field, temperature, pore-pressure, etc). They are typically modeled by a set of partial differential equations (PDEs), to characterize different physics at different parts of the domain. The solution of these equations poses a challenge regarding the ability of the algorithms to handle such interactions and differences in physics in a general and efficient manner.

Multi-domain problems usually derive from a *bigger* problem definition that can be splitted into *smaller* independent problems. Some common reasons that lead to such a split is to end up with subdomains with simpler shape (geometry), or to separate areas on the original problem with different physics. These subdomains may be coupled through interface operators, depending on the physics of the adjacent or overlapping subdomains on their common interface. Performing such a separation into concrete, independent subdomains, provides the opportunity for faster grid generation (smaller mesh) and parallel solving, which usually means faster overall solution of the initial problem.

In the past, due to the lack of computational capabilities, many attributes of important MDMP problems were either ignored or heavily approximated. However, with current capabilities many of these attributes can be modeled more accurately. This leads to a better understanding of the causes and consequences of natural phenomena and to

more economical and safer products with a deeper insight into the performance of their design.

2.1 The finite element method

The finite element method (FEM) [7] is a numerical technique that applies to boundary value PDE problems. By subdividing the original problem into small simple areas called finite elements, it approximates complex equations into many simpler element equations and combines them to calculate an approximate solution. Using variational methods it can minimize the error w.r.t. the final result and produce a stable solution.

One important feature of the method is that it allows control of the precision of the solution in particular subdomains of interest w.r.t. the whole domain, especially when its characteristics change. Therefore it is commonly embedded inside complicated domains like physical system simulations (e.g. crash simulation, weather prediction). A more detailed discussion can be found in [8].

2.1.1 General principles

Breaking the whole domain into smaller pieces allows a more accurate representation of a complex geometry from one or more materials with different properties, while maintaining the representation of the total problem.

A typical application of the method consists of two steps:

1. division of the whole problem's domain into subdomains, where a set of element equations represents the problem's equation.
2. final calculation of the solution by combining systematically all element equations together. There are known iterative techniques that calculate the final solution of a global system of equations like this, starting from an initial rough solution of the original problem.

The first step approximates the original complex equations (often PDEs) with simpler local element equations. It constructs an integral of the inner product of the residual and

the weight functions and set the integral to zero. That procedure fits trial functions into the PDE to minimize the approximation error. These trial functions cause a residual (error) which is projected through some polynomial weight functions.

The process approximates the PDE locally with a set of algebraic or ordinary differential equations for steady state and transient problems respectively. These equation sets can be linear or nonlinear. We solve algebraic equation sets using numerical linear algebra methods and ordinary differential equation sets using standard numerical integration methods (Euler, Runge-Kutta).

The second step transforms the coordinates of the subdomains' local nodes to the domain's global nodes in order to generate a global system of equations. This transformation applies in relation to the reference coordinate system.

A general form of the finite element method is characterized by the following process:

1. Choose a grid for the problem's domain. The grid consists of triangles or curvilinear polygons in the case of 2D domains or of tetrahedral-shaped finite elements in the case of 3D domains.

2. Choose basis functions, piecewise linear basis functions or piecewise polynomial basis functions.

Chapter 3

FEniCS project

The FEniCS project [1] is a collection of open software tools specialized on automated efficient solution of differential equations.

This section is a brief overview of the FEniCS components we are interested into. A more detailed presentation for the whole project can be found on the FEniCS book [1] available online from the project's main webpage.

3.1 Choosing FEniCS as base platform

The reason we decided to base our platform on the FEniCS project rather than other known solutions is that it is an open source cross platform solution that comes with a number of features useful for the computational scientist.

The most important of these, that automate the assembly and solving phase, are the following:

Dolfin [9] the main frontend for the user that abstracts the implementation details of the individual components of FEniCS, while providing a feature-full API to the user.

UFL [10] the Unified Form Language, a near-mathematical notation expression for PDEs (we discuss more about it in Chapter 3.3).

A variety of finite element spaces for the user to choose depending on the problem, for both 2D and 3D domains.

A variety of linear algebra backends that support different families of solvers, most of them configurable by the user depending on her needs.

In Figure 3.1 we can see a diagram of FEniCS structure. The development of our platform focuses mostly on the Dolfin user interface of FEniCS and the ability to support new external libraries in a transparent way.

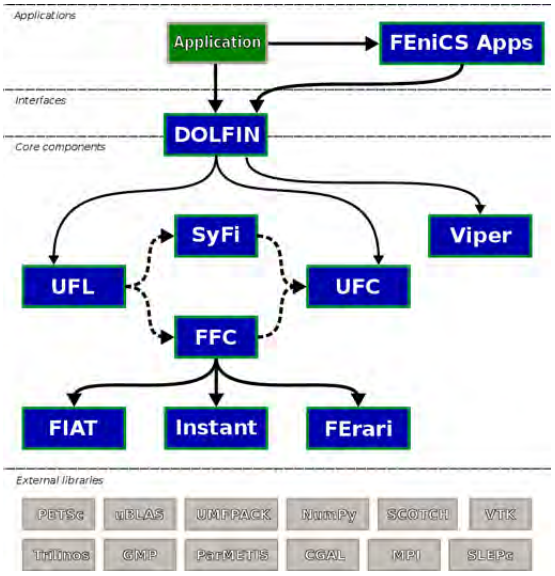


FIGURE 3.1: FEniCS structure [1, p.172]

3.2 The Dolfin library

Dolfin [9] is a C++/Python library that functions as the main user interface of FEniCS. A large part of the functionality of FEniCS is implemented as part of it. It provides a problem solving environment for models based on PDEs. It implements core parts of the functionality of FEniCS, including data structures and algorithms for computational meshes and finite element assembly. To provide a simple and consistent user interface, Dolfin wraps the functionality of other FEniCS components and external software, and is responsible for the correct communication among them.

Name	Symbol
Bubble	B
Crouzeix–Raviart	CR
Discontinuous Lagrange	DG
Discontinuous Raviart-Thomas	DRT
Lagrange	CG
Nedelec 1st kind H(curl)	N1curl
Nedelec 2nd kind H(curl)	N2curl
Quadrature	Q
Raviart–Thomas	RT
Real	R

TABLE 3.1: List of finite elements fully supported by DOLFIN 1.4.

3.2.1 Finite elements and meshes

FEniCS provides an extensive library of finite elements. Table 3.1 lists the supported finite elements.

It also provides fully distributed simplex meshes in one (intervals), two (triangles) and three (tetrahedra) space dimensions. Meshes may be refined adaptively, and there is support for parallel computing through mesh partitioning. Figure 3.2 shows an example.

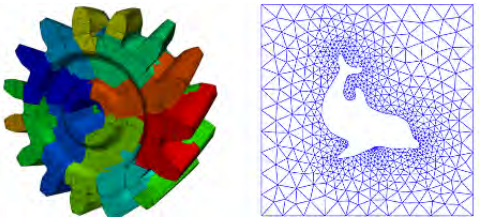


FIGURE 3.2: Meshes [1, p. 214, 205]

3.3 The Unified Form Language (UFL)

UFL [10] is one of the core components of the FEniCS framework. It is a domain specific language for the declaration of finite element discretization of variational forms and functionals, expressing nonlinear PDEs and automatic differentiation of expressions and forms. More precisely, the language defines a flexible user interface for defining finite element spaces and expressions for weak forms in a notation close to mathematical notation. It can handle complicated equations efficiently and differentiation of expressions and forms is integrated in the language.

Weak formulations are an important tool for the analysis of mathematical equations. They permit the transfer of concepts of linear algebra to solve problems in other fields, such as partial differential equations. In a weak formulation, an equation is no longer required to to be satisfied pointwise and has instead weak solutions only with respect to certain "test vectors" or "test functions" [11, p. 24]. This is equivalent to formulating the problem to require a solution in the sense of a distribution.

User friendly notation and support for rapid development are core values in the design of UFL. Having a notation close to the mathematical abstractions allows expression of particular ideas more easily, which can reduce the probability of bugs in user code.

3.4 Notation example

Using the UFL notation one can specify finite element variational problems in near-mathematical notation directly in their programs' source code. One example from the FEniCS book is the variational problem for the Poisson equation [1, p. 3] below:

$$\underbrace{\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x}_{a(u,v)} = \underbrace{\int_\Omega f v \, \mathrm{d}x}_{L(v)} \quad \forall v \in V.$$

Listing 3.1 shows how the formula translates to the FEniCS notation:

```
1 u = TrialFunction(V)
2 v = TestFunction(V)
3
4 a = dot(grad(u), grad(v))*dx
5 L = f*v*dx
```

Listing 3.1: PDE definition in FEniCS UFL notation

The code to automatically solve the above variational problem is illustrated in listing 3.2 below:

```
1 u = Function(V)
2 solve(a == L, u, bc)
```

Listing 3.2: PDE solving in FEniCS UFL

3.5 Linear algebra backends

FEniCS provides unified access to a range of high performance linear algebra libraries through a common wrapper layer. Currently supported linear algebra backends include PETSc [12], Trilinos/Epetra [13], uBLAS [14] and MTL4 [15]. The user can easily switch from one backend to any other by changing the value of a parameter in her code. Some backends also offer support for parallel computing (PETSc, Epetra).

Each backend offers a wide range of tools to work with, including vectors, dense and sparse matrices, direct and iterative linear solvers and eigenvalues solvers. Dolfin defines a simple yet powerful and consistent common interface to support various linear algebra backends.

In particular it defines the abstract base classes GenericTensor, GenericMatrix and GenericVector and uses them throughout the user interface and library.

Part II

The Platform

Chapter 4

FEniCS Extensions

Our platform utilizes and extends the Python user interface of the FEniCS Dolfin library. The reason behind our preference in Python over C++ is clearly practical. Its syntax is closer to UFL syntax and is less time consuming to experiment with due to its scripting nature. The platform is based on FEniCS 1.3.

4.1 Design

We focus on multi-domain multi-physics (MDMP) problems modeled with partial differential equations (PDEs). Every new feature is implemented on top of the existing functionality, either as a new Python module using the available data structures and classes, or as an external dynamically shared C++ library, wrapped as a Python module using SWIG [16].

Our goal is to design and offer an enhanced meta-computing environment based on scripting languages (Python) and their practices, that facilitates the numerical solution of PDEs associated with MDMP mathematical models. To accomplish that, we exploit state of the art numerical solvers offered by the supported FEniCS linear algebra backends.

The platform aims to cover a wide range of problems, following a generic design that can support arbitrary shapes (rectangular or curvilinear) for domains and interfaces between them, for both 2D and 3D geometries.

Apart from the new features and methodologies the platform offers, another critical decision during the development phase was to keep compatibility with existing user codebase. To eliminate the chance of breaking any existing functionality we keep the official release of FEniCS unmodified, putting all the new functionality on external Python modules as discussed earlier.

Two cases of problems with great interest are problems with different elliptic differential operators on different subdomains as well as problems with different PDE discretization and solving modules on different subdomains. FEniCS already supports independent subdomain definitions; the platforms honors the existing infrastructure and builds upon it.

4.2 Supported methodologies

There are two new methodologies integrated to our platform, that can be used directly with any existing type of MDMP problem, as far as it conforms to the mathematical model behind them.

One is a hybrid stochastic/deterministic Monte Carlo-based approach [17] to estimate the boundary values over a subdomain's interface, as presented in Chapter 5. The other is an overlapping domain decomposition method known as the classical alternating Schwarz method [6, chapter 2.1], which is discussed in detail in Chapter 6.

These two methodologies are orthogonal to each other. The Monte Carlo approach can be combined with any supported linear algebra solver in order to provide a fully hybrid stochastic/deterministic PDE solver for subdomains of the original domain. The alternating Schwarz method can use any of the supported linear algebra solvers for each subdomain and offers communication / relaxation at subdomain interfaces.

Chapter 5

Hybrid Stochastic/Deterministic PDE Solving Method

The common approach to compute boundary values, is a fully deterministic computation on each boundary point, by evaluating a function over all boundary points. Another rare, yet deterministic approach is for the user to manually assign values at boundary points. This chapter describes in detail the hybrid stochastic/deterministic Monte Carlo-based approach [4, 5] to compute boundary values.

Monte Carlo methods have the capability to provide approximate solutions to a variety of mathematical and engineering problems, by performing statistical sampling experiments. Observing the characteristics and behavior of the results, they are capable of calculating approximations to PDE solutions. Despite they are generally considered as methods of last resort, ideally suitable only for problems either in high dimensions or very complex geometries, they have been commonly used for many important problems. Apart from that, their inherent parallelism makes them a suitable candidate for modern hardware devices such as GPGPUs and FPGAs.

5.1 Theory

Given a domain Ω with boundary $\vartheta\Omega$, a PDE which holds inside Ω and a subdomain of interest $D \subset \Omega$ with boundary Γ internal to Ω, as shown in figure 5.1 the main steps of a hybrid stochastic/deterministic solver based on the Monte Carlo method are:

Stochastic preprossessing: A number of Monte Carlo-based walks on spheres inside Ω decouples the original PDE problem into a set of independent PDE subproblems, in order to estimate Γ.

Interpolation smoothing: Interpolation uses the computed Monte Carlo approximations at selected points on the interface to provide accurate-enough boundary conditions at all points of the interface.

Deterministic solving: Given the value estimations at the subdomain interface, apply a finite element solver for independently computing the PDE solution within the subdomain.

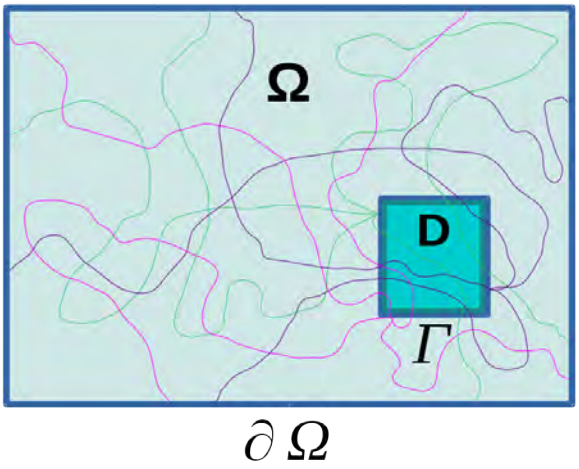


FIGURE 5.1: Random walks inside Ω and D for three different points on Γ (green, purple, magenta)

More information about the original implementation and the theoretical background can be found in [4, 5].

5.2 Implementation

An original, prototype implementation [4] focuses on the Poisson equation, narrowed on the unit square or unit cube for 2D and 3D problems respectively. It also utilizes a laplace solver from the deal.II [2, 3] software library for the step of deterministic solving.

5.2.1 External C++ library

For the purposes of our platform, in order to deliver support for different kinds of problems, a variety of state of the art numerical solvers and arbitrary boundary geometries, we stripped down the original implementation, keeping just the stochastic preprocessing step. The two missing steps (interpolation, solving) were replaced with their respective alternatives as offered by the supported FEniCS linear algebra backends. A big advantage that comes with this decision is the decoupling of the computations for the stochastic step from the actual solving algorithm. It further allows us to run the stochastic preprocessing step in any device type (FPGA, GPGPU) and optimize it independently in order to benefit from the inherent parallelism of the Monte Carlo approach.

This stripped version of the library defines a MC object and calls the MC::monte_carlo() method which takes the coordinates of the boundary nodes as input and outputs the estimation value for each boundary node.

This version offers a stable parallel implementation using POSIX threads [18] to run on CPU, and an OpenCL [19] implementation which can run on every OpenCL capable device. Support for arbitrary boundary geometries is in experimental state; this version supports only rectangular domains.

Listing 5.1 shows the base class Problem for the Poisson equation:

```
1  template <int dim>
2  class Problem {
3  private:
4      std::shared_ptr<const dolfin::Expression> f_expr, q_expr;
5  public:
6      double D[dim];
7      Problem(const double *D,
8              std::shared_ptr<const dolfin::Expression> _f = 0,
9              std::shared_ptr<const dolfin::Expression> _q = 0) :
10         f_expr(_f), q_expr(_q)
11     { for (int i=0; i<dim; ++i)  this->D[i] = D[i]; }
12
13     double f(const double *_x) {
14         dolfin::Array<double> values(1);
15         dolfin::Array<double> x(dim,const_cast<double *>(_x));
16         f_expr->eval(values,x);
17         return values[0];
18     }
19
```

```
20      double q(const double *_x) {
21          dolfin::Array<double> values(1);
22          dolfin::Array<double> x(dim,const_cast<double *>(_x));
23          q_expr->eval(values,x);
24          return values[0];
25      }
26  };
```

Listing 5.1: C++ base class Problem for problem definition

Subdomain D holds the lengths per dimension for the domain Ω, where q() and f() are user defined functions that verify the Poisson's equation $\nabla^2 q(x) = f(x)$. The parameter x holds the coordinates of a 2D or 3D point to evaluate. ∇ is the Laplace operator [20].

Both q() and f() expressions are constructed using the FEniCS API. Listing 5.2 shows a definition in UFL notation.

```
1  def Laplacian(expr,x,y):
2      dxexpr = diff(expr,x)
3      dx2expr = diff(dxexpr,x)
4
5      dyexpr = diff(expr,y)
6      dy2expr = diff(dyexpr,y)
7
8      dx2dy2expr = dx2expr + dy2expr
9      return dx2dy2expr
10
11 x = variable(Expression("x[0]"))
12 y = variable(Expression("x[1]"))
13 f = (x)*(x-1)*(y)*(y-1)
14 q = -Laplacian(f,x,y)
```

Listing 5.2: UFL definition of the same Poisson equation

For the OpenCL version of the algorithm the platform provides two skeleton OpenCL kernels, one for 2D and one for 3D problems. The platform generates the definitions of q() and f(), appends the proper OpenCL kernel code that uses them, compiles and runs the generated code. The f and q expressions in listing 5.2 will generate the following code in 5.3:

```
1  inline double CPP_CODE_Q(const double *x) { return -2*(x*(x-1) + y*(y-1)); }
2  inline double CPP_CODE_F(const double *x) { return x*(x-1)*y*(y-1); }
```

Listing 5.3: C++ prototype of the montecarlo() method

The C++ prototypes of the montecarlo() method are shown in listing 5.4:

```
1  // multithread cpu version
2  std::vector<double>
3  montecarlo(double *D, int dim, double* node_coord, int nof_nodes,
4             std::shared_ptr<dolfin::Expression> f,
5             std::shared_ptr<dolfin::Expression> q);
6
7  // parallel OpenCL version
8  std::vector<double>
9  montecarlo(double *D, int dim, double* node_coord, int nof_nodes,
10            const std::string &f,
11            const std::string &q);
```

LISTING 5.4: C++ prototype of the montecarlo() method

The result (estimated values) is returned to the caller for further processing.

5.2.2 Python wrapper module

The Python module provides a montecarlo() wrapper method that calls the external C++ library through a wrapper layer generated by SWIG [16].

The wrapper method takes the same arguments with the DirichletBC class, plus the size (per dimension) of the domain. Using the DirichletBC methods we obtain the points on the boundary and call the external C++ library on them, with the appropriate parameters (the user defined expressions). The C++ call returns the estimated values of all boundary points (nodes) and the wrapper assigns them to a new DirichletBC object (actually to its vector attribute). Finally the wrapper method returns this new DirichletBC object which can be used anywhere in the rest of the program.

Listing 5.5 shows the implementation of the montecarlo() wrapper method that calls the SWIG generated wrapper layer:

```
1  from fenics import *
2  import _hybridmc as core
3  import hmc_toolbox as tools
4  import numpy as np
5
6  def montecarlo(self,V,interface,**kwargs):
7      dims = kwargs.get('Omega')
8      bc = DirichletBC(V,1.0,interface)
9      coords, keys = tools.get_boundary_coords(bc)
```

```
10      dim = len(dims)
11      nof_nodes = len(coords)/dim
12      D = np.array(dims, dtype=np.float_)
13      node_coord = np.array(coords, dtype=np.float_)
14
15      f, q = kwargs.get('f'), kwargs.get('q')
16      if not kwargs.get('OpenCL',False):
17          f, q = Expression(f), Expression(q)
18      value = core.montecarlo(D,dim,node_coord,nof_nodes,f,q)
19      est = Function(V)
20      est.vector()[keys] = value
21      mcbc = DirichletBC(V,est,interface)
22      return mcbc, est
```

Listing 5.5: Definition of montecarlo() method

There are some explicit data conversions from Python datatypes to NumPy [21] datatypes as we need to guarantee that our data lie in contiguous memory. NumPy is also used internally from the FEniCS Python interface.

With the proper configuration, SWIG generates the appropriate wrapper code to automatically convert an std::vector to a NumPy array and vice versa.

We also introduce an extra Python module that implements a few helper functions that glue different components of FEniCS together and hide the unnecessary details from the programmer. It also simplifies the implementation of the web services support as we discuss in chapter 7.

5.3 Example

A toy example that illustrates how one can use this new method is shown in listing 5.6:

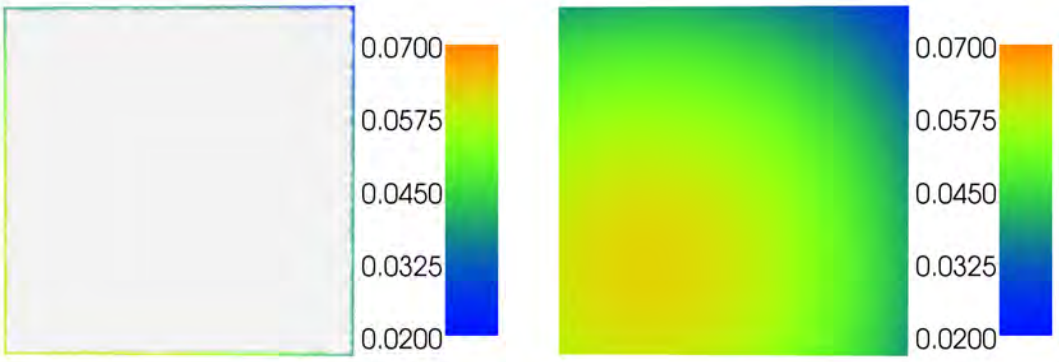
```
1  from dolfin import *
2  import hybridmc as hmc  # the platform's Python module
3
4  def onbc(x,on_boundary):
5      return on_boundary
6
7  def mc_test(Omega, Subdomain):
8      x = variable(Expression("x[0]"))
9      y = variable(Expression("x[1]"))
10     expr = (x)*(x-1)*(y)*(y-1)
11
```

```
12      mesh = Mesh(SubDomain,128)
13      V = FunctionSpace(mesh,'Lagrange',1)
14
15      u, v = TrialFunction(V), TestFunction(V)
16      f = -Laplacian(expr,x,y)
17      a = inner(grad(u), grad(v))*dx
18      L = f*v*dx
19
20      # get expression as string
21      f_expr = hmc.tools.cppcode(expr,x,y)
22      q_expr = hmc.tools.cppcode(f,x,y)
23      mcbc, est = client.montecarlo(V, onbc, OpenCL=True, Omega=Omega,
24                                    f=f_expr, q=q_expr)
25      sol_mc = Function(V)
26      solve(a==L,sol_mc,[ mcbc ])
27
28      plot(mcbc,title='monte carlo bc')
29      plot(sol_mc,title='monte carlo solution',scale=0.0)
30      interactive() #hold plots
31
32  if __name__ == '__main__':
33      Omega = [ 1., 1. ]
34      SubDomain = Rectangle(.4, .8,
35                            .4, .8)
36      client = hmc.LocalClient()
37      mc_test(Omega, Subdomain)
```

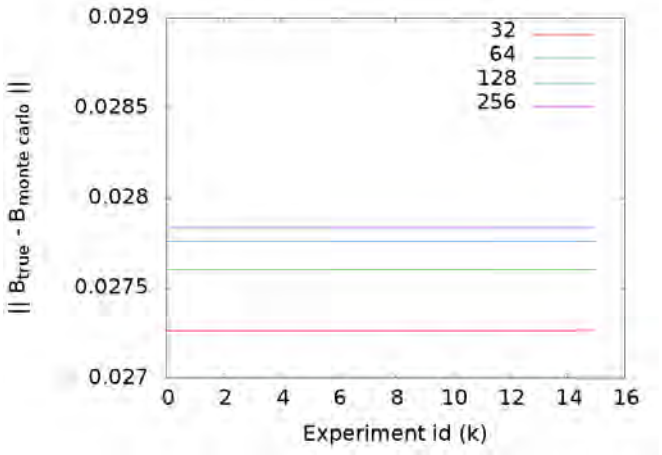
Listing 5.6: Example of montecarlo() method in user code

The client object at line 36 above provides the local/remote functionality of the method. We discuss more about web services and client objects in chapter 7.

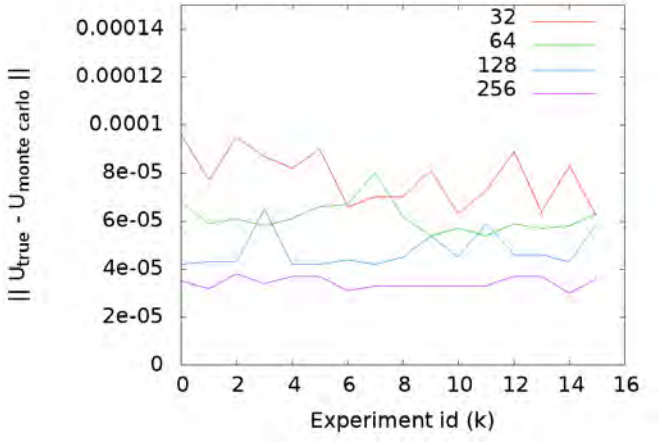
Figure 5.2 shows (a) the estimated values of Γ using the Monte Carlo method, (b) the solution of the hybrid stochastic/deterministic Monte Carlo-based solver, (c) the estimation error w.r.t. the deterministic approach and (d) the solution error w.r.t. a fully deterministic solver for different mesh resolutions.



(A) MonteCarlo estimation on Γ (B) Solution of a Monte Carlo-based solver



(C) Error w.r.t. the deterministic boundary estimation



(D) Solution error w.r.t. a fully deterministic solver

FIGURE 5.2: Plots from the example code in listing 5.6

Chapter 6

Schwarz Alternating Method

6.1 Theory

The theory covered in this chapter is a brief presentation of Cai [6, chapter 2], where the author explains the mathematical formulation of overlapping domain decomposition methods in more detail.

6.1.1 Overlapping domain decomposition

Overlapping domain decomposition methods [6] are efficient and flexible. Such methods are inherently suitable for parallelizing the solution of partial differential equations (PDEs), where the methods of concern are based on a physical decomposition of a global solution domain. The global solution to a PDE is then achieved by solving the smaller subdomain problems collaboratively and then combining the individual solutions.

6.1.2 Classical alternating Schwarz method

The classical alternating Schwarz method demonstrates the basic idea of overlapping domain decomposition methods.

Considering an example domain Ω created from the union of a circle Ω_1 and a rectangle Ω_2, as figure 6.1 shows and a specific Poisson equation, the problem can be written as:

$$-\nabla^2 u = f \text{ in } \Omega = \Omega_1 \cap \Omega_2,$$
$$u = g \text{ on } \vartheta\Omega.$$

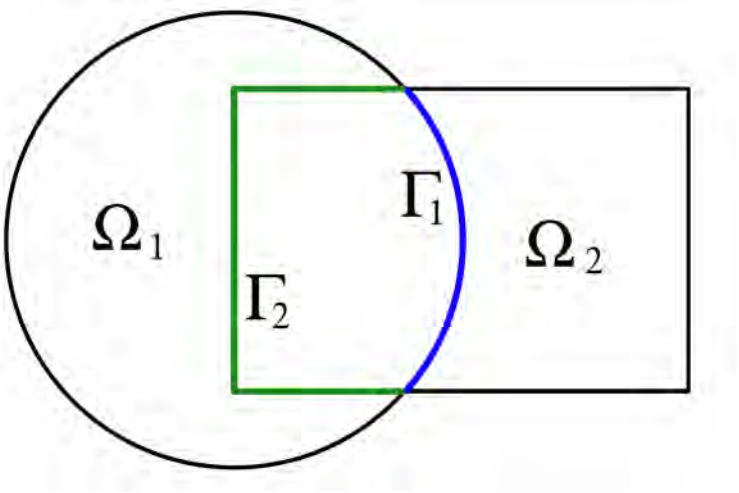


FIGURE 6.1: Solution domain for the classical alternating Schwarz method. [6, p. 3]

Any part of a subdomain boundary $\vartheta\Omega_i$ which is not part of the global physical boundary $\vartheta\Omega$, is referred in the bibliography as *artificial internal boundary*. In figure 6.1, we see that Γ_1 is the artificial internal boundary of subdomain Ω_1, and Γ_2 is the artificial internal boundary of subdomain Ω_2.

In order to solve separately the PDE on each subdomain, Schwarz proposed the utilization of analytical solution methods in an iterative procedure that finds the approximate solution in the entire composite domain Ω. Expressing the approximate solution in subdomain Ω_i as u_i^n and the restriction of f as f_i, we can start with an initial guess u_0 and iterate over the previous iteration in order to find more approximate solutions u^1, u^2, and so on, for n iterations.

During each iteration, for each subdomain i, we solve the PDE restricted to Ω_i using the solution from the neighboring subdomain on Γ_i from the previous iteration. Considering the domains in figure 6.1, we have:

$$\begin{aligned} -\nabla^2 u_1^n &= f_1 \text{ in } \Omega_1, \\ u_1^n &= g \text{ on } \vartheta\Omega_1 \backslash \Gamma_1, \\ u_1^n &= u_2^{n-1} \mid \Gamma_1 \text{ on } \Gamma_1. \end{aligned} \qquad \begin{aligned} -\nabla^2 u_2^n &= f_2 \text{ in } \Omega_2, \\ u_2^n &= g \text{ on } \vartheta\Omega_2 \backslash \Gamma_2, \\ u_2^n &= u_1^n | \Gamma_2 \text{ on } \Gamma_2. \end{aligned}$$

We can start the method by first solving either the Ω_1 or the Ω_2 subdomain without any noticeable effects on the convergence.

At the end of each iteration, we need to update the artificial Dirichlet conditions on every associated Γ_i with values from the new solution. This is necessary while converging towards the final accurate solution because as a rule, the values on Γ_i converge to their final values after each step.

6.1.3 Additive Schwarz method

The additive Schwarz method is another variant of overlapping domain decomposition methods, which inherently promotes parallel computing. Its difference from the classical method lies in the way the artificial Dirichlet condition ($\tilde{g}$ *notation below*) is updated on Γ_i, i.e. the n-th solution uses the solutions from all the neighboring subdomains from the previous step ($u_i^n = \tilde{g}^{n-1}$). Therefore, at each step the subdomain solutions in the additive Schwarz method are independent and can be carried out in parallel.

Although the inherent parallelism of the additive Schwarz method, it should be noted that its convergence properties are inferior to those of the multiplicative Schwarz method. The additive Schwarz method requires roughly 2x iterations to converge compared with other Schwarz methods.

6.2 Implementation

We implement the additive Schwarz method and use it as a high level solver for MDMP problems. In contrast with the Monte Carlo method in chapter 5, the code is written purely in Python; we do not offer a C++ API for this feature.

Figure 6.2 shows the control flow of the algorithm.

6.2.1 Multi overlapping subdomains

Currently the code supports simple domain overlapping schemes: any particular point contributes to at most two different subdomains. Full support for multi domain overlapping areas as shown in figure 6.3, is in experimental state and may result to convergence

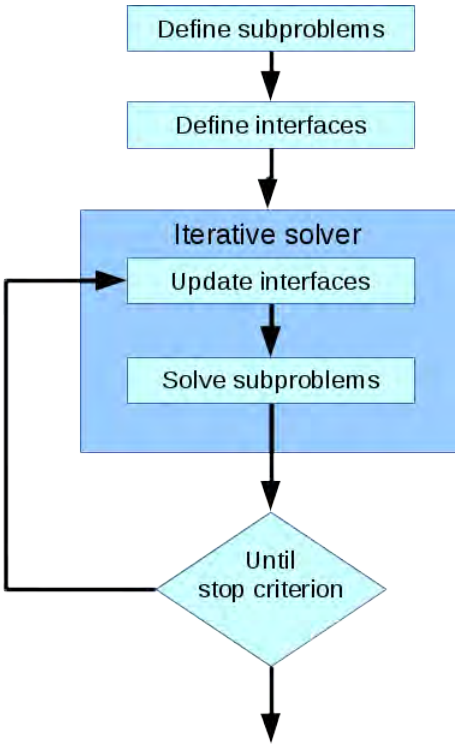


FIGURE 6.2: Control flow of the iterative algorithm

issues, depending on the modeled problem. As the platform evolves, depending on the types of problems we need to solve, different iterative methods may be added.

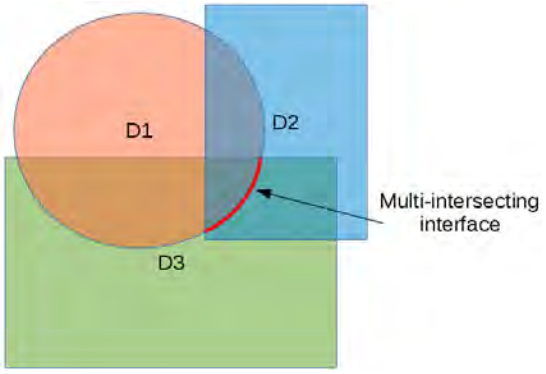


FIGURE 6.3: An example of multi overlapping subdomains

6.2.2 Split problem into files

For the best utilization of our platform, we propose and encourage the use of an organization scheme based on multiple source code files per problem, more precisely one file per subdomain definition.

This organization scheme highlights the independence between domains as distinct programming units. It also allows easy collaboration and sharing between different researchers or research groups and hides problem definition details from the not-interested parties. Another more technical reason is that this subdomain separation to files greatly simplifies the implementation for supporting remote solvers and methods as web services, a direction which is quite attractive due to its performance benefits.

6.2.3 Python module

All underlying datatypes of the base classes are either pure Python or FEniCS objects. There is no dependence from third party software libraries at this level. The Python module consists of two files:

solverconfig.py provides the base classes with sanity checks and the API for the solver to function properly.

solver.py implements the solving routine among a handful of helper functions that simplify the whole process and further sanity checks to ensure the proper setup of the user's problem.

Inside solverconfig.py the module defines the following base classes:

class LogInfo Its purpose is to keep track of the progress of a particular subdomain. The available information can be written to a user defined logfile.

class ConfigCommon Holds separately the configuration of the whole solver. Some of the attributes the user can set are the number of dimensions of the problem, the number of max iterations for the solver, a tolerance value that is used to check for convergence, the filenames of the subdomains which will take part in the solving process, whether the user wants the creation of logfiles and whether they want visual plots of the solutions in each iteration. The class provides some predefined default values for all attributes.

class Config2D Derives from ConfigCommon, with predefined number of dimensions set to 2. Everything else is the same as the parent class.

class Config3D Derives from ConfigCommon, with predefined number of dimensions set to 3. Everything else is the same as the parent class.

class ConfigCommonProblem This is the base class the user extends to define each subdomain. There are three methods that need to be overridden. We discuss them in detail in section 6.2.3.1.

6.2.3.1 Domain API

Each subdomain object that inherits from the ConfigCommonProblem base class must override the following methods. The solver object calls these methods before the actual solving phase begins, in order to gather the appropriate useful information and setup the appropriate data structures for each subdomain.

init() This method holds the UFL [10] definition of the subdomain and sets as class attributes the subdomain's function space, linear and bilinear form of the PDE.

neighbors() It provides information to the solver about the other subdomains this subdomain overlaps with, in order for the solver to automatically update the boundary values after each iteration. It returns a Python dictionary with keys the filename of the neighbor subdomain and as value a method that returns the boolean value True only for the nodes on the common boundary of this subdomain and the neighbor subdomain.

boundaries() It informs the solver about the fixed external boundaries of the subdomain. It returns a Python list of all the subdomain's external boundaries, each element being a DirichletBC object.

6.2.3.2 Iterative solver

The entry point of the iterative solver is the solve() method as defined inside solver.py. It takes as arguments a ConfigCommon object with the configuration of the solving environment (max iterations, tolerance, etc) and a Python list of user defined problem objects, all derived from ConfigCommonProblem base class. After some initial steps (create logfiles, initialize solution vectors, etc), the main solving routine is called, named __solve(subdomains, config).

The main points of interest of the iteration algorithm and two of the helper methods are shown below in listing 6.1:

```
1  def __interpolate_interfaces(subdomains):
2      for subdomain in subdomains:
3          for iface in subdomain.interfaces.itervalues():
4              interpolant = interpolate(iface['solution'],subdomain.trial_space())
5              iface['interpolant'].vector()[:] = interpolant.vector()
6
7  def __solve_iteration(subdomains):
8      for subdomain in subdomains:
9          subdomain.solve()
10
11 def __update_interfaces(subdomains):
12     for subdomain in subdomains:
13         for iface in subdomain.interfaces.itervalues():
14             iface['previous'].vector()[:] = iface['current'].vector()
15             iface['bc'].apply(iface['current'].vector())
16
17 def __solve(subdomains,config):
18     iteration = 0
19     iterate = True
20     while iterate:
21         iteration += 1
22
23         __interpolate_interfaces(subdomains)
24         __solve_iteration(subdomains)
25         __update_interfaces(subdomains)
26
27         if config.show_solution_plots:
28             for subdomain in subdomains:
29                 plot(subdomain.solution(),title=subdomain.name)
30         for subdomain in subdomains:
31             if stop_criterion(config,subdomain,iteration):
32                 iterate = False
33
34     return [ subdomain.solution() for subdomain in subdomains ]
```

Listing 6.1: Core code of the iterative algorithm routine

After each iteration, for each subdomain solution, the algorithm checks a set of termination criteria in the following order that may terminate the solving process:

1. If the exact solution is known, check for convergence w.r.t. the user defined tolerance value.

2. If the errornorm of the current and previous iteration is below a user defined tolerance value.

3. If the max iterations limit as defined by the user is reached. This is also the only case of failure.

Below, in listing 6.2, we see the structure of the stop_criterion() method:

```
1  def stop_criterion(config,subdomain,iteration):
2      converged = True
3      for iface in subdomain.interfaces.itervalues():
4          if not __stop_criterion(config,iface,iteration):
5              converged = False
6      return converged
7
8  def __stop_criterion(config,iface,iteration):
9      loginfo = iface['log']
10     x = iface['current']
11     x_prev = iface['previous']
12     x_exact = iface['exact'] if 'exact' in iface else None
13
14     if x_exact:
15         err_exact = errornorm(x_exact,x)
16         n = norm(x_exact)
17         if n != 0:
18             err_exact /= n
19
20     err_prev = errornorm(x,x_prev,degree_rise=degree_rise)
21
22     if x_exact and err_exact <= config.tol_exact:
23             print "***  matched exact solution  ***"
24             return True
25     if iteration != 1 and err_prev <= config.tol_prev:
26         print "***  no change between iterations  ***"
27         return True
28     if iteration > config.max_iterations:
29         print "***  max iterations limit reached  ***"
30         return True
31     return False
```

Listing 6.2: Implementation of the stop_criterion() method

Note that in order for the stop_criterion() method to terminate the algorithm, all subdomains must converge for either of the two first criteria. The third criterion is common for all subdomains.

The solver keeps logfiles for each boundary interface between all overlapping subdomains. They keep track of the progress per iteration in a column based format which is suitable to use as input to Gnuplot [22].

6.3 Example

For example a skeleton file (circle2D_1.py) with the definition (in Python) for the circle subdomain in figure 6.1 can be the following as shown in listing 6.3:

```
1  # user defined methods
2  def OveralappingWithOther():            pass
3  def getOrCreateMesh():                  pass
4  def userDefinedUFL():                   pass
5  def userDefinedBoundaryCondition():     pass
6
7  # skeleton example
8  def ExtBC(x,on_boundary):
9      return on_boundary and not OveralappingWithOther()
10
11 def ExtIface(x,on_boundary):
12     return on_boundary and OveralappingWithOther()
13
14 class Problem(ConfigCommonProblem):
15     def init(self,*args,**kwargs):
16         mesh = getOrCreateMesh(*args,**kwargs)
17         self.V = FunctionSpace(mesh,'Lagrange',1)
18         self.a, self.L = userDefinedUFL(V)
19
20     def neighbors(self):
21         interface = {}
22         interface['rectangle'] = ExtIface
23         return interface
24
25     def boundaries(self):
26         bc = DirichletBC(self.V, userDefinedBoundaryCondition(), ExtBC)
27         return [ bc ]
```

Listing 6.3: Common skeleton code example for subdomain definitions

The skeleton definition is abstract to the geometry and number of dimensions of the subdomain. That means that the same skeleton code from listing 6.3 can be used to define the rectangle subdomain as well.

Given two defined subdomains in files circle2D_1.py and rectangle2D_1.py, the driver code that solves them looks like this in listing 6.4:

```
1  from dolfin import *
2  import solverconfig
3  import solver
4
5  import circle2D_1 as circle
6  import rectangle2D_1 as rectangle
7
8  cp = circle.Problem()
9  rp = rectangle.Problem()
10 subdomains=[ cp, rp ]
11
12 config = solverconfig.Config2D()
13 solver.solve(subdomains=subdomains,config=config)
14
15 # keep plots on screen
16 interactive()
```

Listing 6.4: Code example that solves two overlapping subdomains

The source code of a fully working 3D example of the iterative solver, along with some plots of the solutions during iterations, are provided in appendix A.1.

Chapter 7

Web Services

This chapter explores the idea of using solvers and methods offered by remote machines in a transparent and abstract way from within the platform.

Beyond acting as a client using services from remote nodes, the platform can also act as a server, by advertising its capabilities as web services, through the SOAP protocol [23] specification and WSDL language [24].

7.1 About SOAP

The Simple Object Access protocol (SOAP) is an XML-based protocol specification for exchanging structured information through web services over computer networks, that relies on other application layer protocols, such as HTTP (Hypertext Transfer Protocol) or SMTP (Simple Mail Transfer Protocol), for message transmission. SOAP can be a core messaging framework for web services. It is also independent from any programming model, thus it can operate on a wide range of possible use cases.

SOAP consists of the following key parts:

- definition of the message structure and means to process it (envelope)
- encoding rules allowing the expression of application-defined datatypes
- conventions for procedure call and response representations

For example, when an application sends a SOAP message to a server asking for a web service (e.g. access to a database) with the parameters for a search, the server returns an XML-formatted response with the resulting data, which the application can consume directly.

7.2 About WSDL

The Web Services Description Language (WSDL) is an XML-based interface definition language, used to describe the functionality a particular web service has to offer. It also provides information of how the service should be called in terms of expected parameters and returned data structures.

In WSDL a reusable binding associated with a network address defines an endpoint, whereas a network service is merely a collection of endpoints (ports). Messages describe the data being exchanged where endpoints describe and supported operations.

WSDL and SOAP are often used in combination in order to implement web services. An application (client) can connect to a web service, determine what operations are available by querying the WSDL descriptor and then use SOAP to actually use one of them.

7.3 Implementation

The implementation consists of a server module that advertises the available methods as web services through a WSDL file, and a client module that calls services offered from the local server or any remote machine, by parsing the appropriate WSDL descriptor file each remote machine provides.

7.3.1 Server

There are many Python frameworks to choose from when building a wed service. We based our server side implementation on Spyne [25], a Python RPC toolkit that facilitates exposing online services that have a well-defined API using multiple protocols and transports.

Spyne aims to simplify the development of remote procedure call APIs and the procedure to expose web services using multiple protocols and transports. In other words, Spyne is a framework for building distributed solutions that strictly follow the MVC pattern [**?**], where Model = spyne.model, View = spyne.protocol and Controller = user code. Spyne comes with the implementations of popular transport, protocol and interface document standards along with a well-defined API that lets you build on existing functionality. Spyne currently supports the WSDL 1.1 interface description standard, along with SOAP 1.1.

A web service code consists of methods that the developer provides and wishes to expose to the web. They are regular Python functions that do not need to use any specific API or adhere to any specific convention. A full documentation among some introductory tutorials can be found at the official documentation of Spyne [25].

Listing 7.1 illustrates a simple server function definition that wraps the hybrid stochastic/deterministic PDE solver method:

```
1  from spyne import Application, rpc, ServiceBase
2  from spyne import Integer, Double, Array
3  from spyne.protocol.soap import Soap11
4
5  import _hybridmc as core
6  import numpy as np
7
8  class MDMPService(ServiceBase):
9      """
10     1. convert the input Python lists to numpy arrays
11     2. call the core method
12     3. return output as Python list
13     """
14     @rpc(Array(Double),Integer,Array(Double),Integer,String,String,Boolean,
15          _returns=Array(Double))
16     def montecarlo(ctx, dims, dim, coords, nof_nodes,f,q,OpenCL):
17         D = np.array(dims, dtype=np.float_)
18         node_coord = np.array(coords, dtype=np.float_)
19         if not OpenCL:
20             f = Expression(f)
21             q = Expression(q)
22         value = core.montecarlo(D,dim,node_coord,nof_nodes,f,q)
23         return value
```

Listing 7.1: Expose montecarlo() method as web service

The deployment of server code in listing 7.1 can be done as shown in listing 7.2:

```
1  from spyne import Application
2  from spyne.server.wsgi import WsgiApplication
3  from wsgiref.simple_server import make_server
4  from mdmp_service import MDMPService
5  import logging
6
7  application = Application([MDMPService], 'spyne.examples.hello.soap',
8                            in_protocol=Soap11(validator='lxml'),
9                            out_protocol=Soap11())
10 wsgi_application = WsgiApplication(application)
11
12 logging.basicConfig(level=logging.DEBUG)
13 logging.getLogger('spyne.protocol.xml').setLevel(logging.DEBUG)
14
15 logging.info("listening to http://127.0.0.1:8000")
16 logging.info("wsdl is at: http://localhost:8000/?wsdl")
17
18 server = make_server('127.0.0.1', 8000, wsgi_application)
19 server.serve_forever()
```

LISTING 7.2: Deployment of the web service

7.3.2 Client

The client side utilizes the Python Suds web service client [26], which is a lightweight soap-based client for Python. It provides an object-like API that can read WSDL files at runtime for encoding/decoding, in order to present an RPC-like interface to soap-based web services.

The primary interface of the platform is the RemoteClient class that utilizes the Suds Client class. When the Client is created, it parses the WSDL and derives a representation which is, in turn, used to provide a service description as well as for message/reply processing.

Listing 7.3 shows the base definition of the RemoteClient class:

```
1  from suds.client import Client
2
3  class RemoteClient(Client):
4      def __init__(self,*args,**kwargs):
5          self.is_local = False
6          Client.__init__(self,*args,**kwargs)
```

Listing 7.3: A simple definition of the RemoteClient object

In order to have a consistent API between local and remote methods, apart from the RemoteClient class, we also define a LocalClient class, as shown in listing 7.4, with the same methods available to the user. The only difference is the internal implementation. In the case of RemoteClient, the input data are sent to the remote server which in turn responds with the output data. All this traffic is transparent to the user who receives the result the same way as if the method had been executed locally.

```
1 class LocalClient():
2     def __init__(self,*args,**kwargs):
3         self.is_local = True
```

Listing 7.4: A simple definition of the LocalClient object

The platform then defines in both RemoteClient and LocalClient classes methods that it is going to support, either as remote or local features respectively. For example, the wrapper routines that implement the hybrid PDE solver method described in chapter 5 as remote or local service, can be observed in listings 7.5 and 7.6 respectively. Notice their similarity; the only change required is the actual call to the underlying wrapper method. LocalClient calls the SWIG generated wrapper, where RemoteClient calls the Spyne wrapper.

inside class RemoteClient:

```
1  def montecarlo(self,V,interface,**kwargs):
2      dims = kwargs.get('Omega')
3      bc = DirichletBC(V,1.0,interface)
4      coords, keys = tools.get_boundary_coords(bc)
5      dim = len(dims)
6      nof_nodes = len(coords)/dim
7
8      D = self.factory.create('doubleArray')
9      D.double.extend(dims)
10     node_coord = self.factory.create('doubleArray')
11     node_coord.double.extend(coords)
12
13     OpenCL = kwargs.get('OpenCL',False)
14     f, q = kwargs.get('f'), kwargs.get('q')
15     wrap_value = self.service.montecarlo(D,dim,node_coord,nof_nodes,f,q,OpenCL)
16     value = np.array(wrap_value.double,dtype=np.float_)
```

```
17
18      est = Function(V)
19      est.vector()[keys] = value
20      mcbc = DirichletBC(V,est,interface)
21      return mcbc, est
```

Listing 7.5: Wrapper for remote monteCarlo() support

inside class LocalClient:

```
1  def montecarlo(self,V,interface,**kwargs):
2      import _hybridmc as core
3
4      dims = kwargs.get('Omega')
5      bc = DirichletBC(V,1.0,interface)
6      coords, keys = tools.get_boundary_coords(bc)
7      dim = len(dims)
8      nof_nodes = len(coords)/dim
9      D = np.array(dims, dtype=np.float_)
10     node_coord = np.array(coords, dtype=np.float_)
11
12     OpenCL = kwargs.get('OpenCL',False)
13     f, q = kwargs.get('f'), kwargs.get('q')
14     if not OpenCL:
15         f, q = Expression(f), Expression(q)
16     value = core.montecarlo(D,dim,node_coord,nof_nodes,f,q)
17
18     est = Function(V)
19     est.vector()[keys] = value
20     mcbc = DirichletBC(V,est,interface)
21     return mcbc, est
```

Listing 7.6: Wrapper for local montecarlo() support

7.4 Example

The user can use the remote procedures in the same way she uses local function calls. Listing 7.4 illustrates part of an example program that compares the values over a boundary computed by the default deterministic approach and the stochastic monte carlo approach. The only part of the user code that changes is the definition of the client object.

```
1  from dolfin import *
2  import hybridmc as hmc
```

```
3
4  if use_remote_client:
5      client = hmc.RemoteClient(wsdl_url)
6      client.set_options(timeout=90)      # ****    IMPORTANT    ****
7  else:
8      client = hmc.LocalClient()
9
10 V = FunctionSpace(mesh,'Lagrange',1)
11 x = variable(Expression("x[0]"))
12 y = variable(Expression("x[1]"))
13
14 Omega = [ 1., 1. ]
15 f = (x)*(x-1)*(y)*(y-1)
16 q = -2*(x*(x-1) + y*(y-1))
17
18 mcbc, _ = client.montecarlo(V, onbc, OpenCL=True, Omega=Omega, f=f, q=q)
19 bc = DirichletBC(V,f,onbc)
20
21 # compare accurate and monte carlo boundary values
22 diff_bc = hmc.tools.bc_errornorm(bc,mcbc)
```

Chapter 8

An Environmental Engineering Application

To demonstrate the potential of the PDE solving philosophy discussed in earlier chapters and the capability of the associated framework implementation to deal with real-world problems, we consider the steady state problem of saltwater intrusion in coastal aquifers.

We need to prevent the contamination of the pumping point with saltwater, in order to keep the well viable. This could happen if the amount of pumped water is greater than the amount of rain water that returns to the aquifer. In this case, due to the higher density of saltwater, the transition zone between saltwater and freshwater moves further inside the aquifer. Our approach is to model the position of the transition zone, in order to find the maximum amount of freshwater we can pump safely per day without risking contamination of the well.

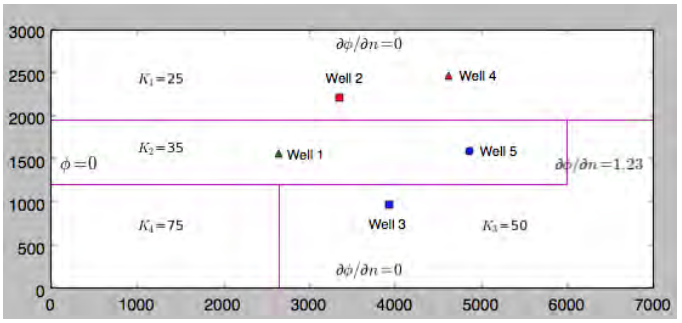


FIGURE 8.1: Kalymnos aquifer.

8.1 Setup

The simplified problem in the 2D space is modelled by the elliptic PDE:

$$\frac{\partial}{\partial x}(K\frac{\partial \phi}{\partial x}) + \frac{\partial}{\partial y}(K\frac{\partial \phi}{\partial y}) + N - Q = 0 \ , \quad (x, y) \in \mathcal{R}, \tag{8.1}$$

where ϕ (m^2) denotes the Struck's flow potential, N (m/day) denotes the total aquifer recharge uniformly distributed over the surface of the aquifer, K (m/day) denotes the hydraulic conductivity and Q (m/day) denotes the total aquifer discharge. Furthermore, let us assume that the rectangular-shaped aquifer $\mathcal{R}$ extends over an area of 7×3 Km, is *heterogeneous* with respect to the hydraulic conductivity, and contains M wells w_i $(i = 1, \ldots, M)$ pumping at Q_i (m^3/day) rates.

The PDE needs to be solved in every iteration step of a stochastic optimization algorithm [27–29], used to optimally control pumping from all active pumping sources (wells) of a coastal aquifer and protect them from salinization. Its solution (flow potential) is being used to locate/determine the interface between salt and fresh water.

The aquifer considered here and depicted in figure 8.1 models a real coastal aquifer located at Bathi area in the Greek island of Kalymnos [30]. The problem is naturally split into four subproblems, as depicted in figure 8.1, that are defined due to the different PDE operator (different hydraulic conductivity Ks in PDE equation 8.1). We further split the right bottom domain into two subdomains, to simplify domain geometry so that it consists of rectangular subdomains only. This results into a total of five subproblems.

For the multi-domain implementation of the problem for Schwarz method, we extend the areas of each subdomain either horizontally or vertically, in order to define a MDMP problem with overlapping subdomains. The left-middle subdomain with hydraulic conductivity K_2 extends for 400 m inside its neighbor subdomains. Similarly, the bottom-left subdomain extends to the right for 400 m, where the middle-right subdomain extends both left and down. This scheme of overlapping regions results into a total of 14 interfaces among the five subdomains.

8.2 Configuration

In our problem the physical parameters have the following values:

- $M = 5$
- $N = 0.03\ m/year$
- $Q_1 = 252\ m^3/day$
- $Q_2 = 450\ m^3/day$
- $Q_3 = 749\ m^3/day$
- $Q_4 = 1045\ m^3/day$
- $Q_5 = 1270\ m^3/day$
- $K_1 = 25\ m/day$
- $K_2 = 35\ m/day$
- $K_3 = 50\ m/day$
- $K_4 = 75\ m/day$

where K_1 - K_4 are the hydraulic conductivity values associated with the four sub-regions of $\mathcal{R}$ (figure 8.1). Moreover, the total discharge rate Q assumes the value

$$Q = \sum_{i=1}^{5} \tilde{Q}_i \delta(x - x_i, y - y_i)$$

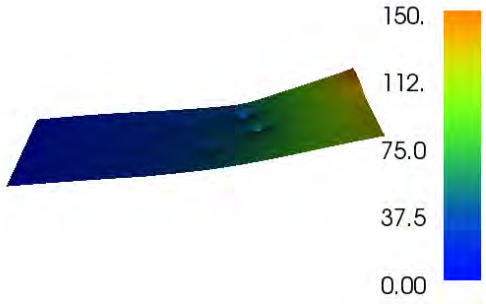
where $\tilde{Q}_i$ denotes the pumping rate Q_i normalized over some elemental area and $\delta(x - x_i, y - y_i)$ denotes the Delta function. Finally, Dirichlet boundary condition ($\phi(0, y) = 0$) is assumed on the left (coastline) edge, while, on all other edges, Neumann boundary conditions are imposed, as shown in figure 8.1.

8.3 Results

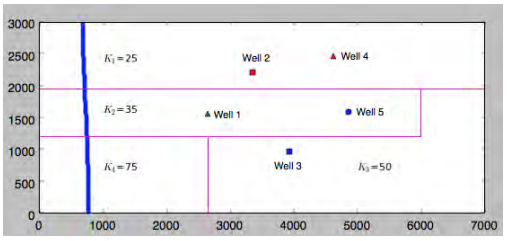
Using the Schwarz method to solve the problem, we get the following results as shown in the following figures. The configuration and source code of the execution can be found on appendix A.2.

Figure 8.2a depicts the computed flow potential using the plotting mechanism from FEniCS [1] Figure 8.2b shows the interface between salt and fresh water is being algorithmically determined in the sequel, using an external python script that utilizes the Matplotlib library [31].

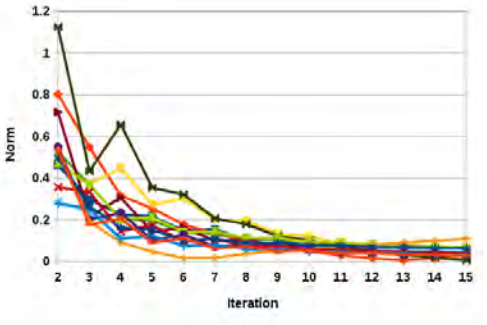
Figures 8.2c and 8.2d show the convergence of the method involving 5 subproblems with a total of 14 interfaces.



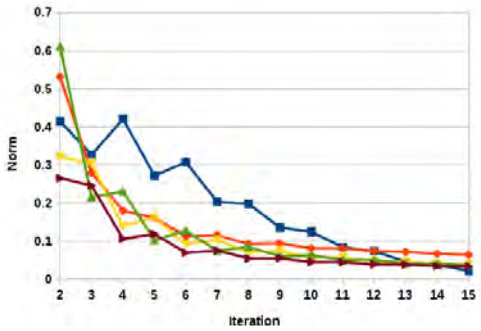
(A) Kalymnos aquifer flow potential.



(B) Interface location between salt and fresh water.



(C) Interface convergence w.r.t. the norm of relative differences in successive iterations on each of the 14 interfaces.



(D) Solution convergence w.r.t. the norm of relative differences in successive iterations on each subproblem.

FIGURE 8.2: Results and convergence for 15 iterations of the Schwarz method.

Chapter 9

Conclusion

We have developed a software platform for MDMP PDE problems, with convenient Application Programming Interfaces and applied it for the effective numerical solution of a practical problem in environmental engineering. Our scheme shows that the metacomputing paradigm for solving composite MDMP problems on state-of-the-art platforms is a very promising approach. It allows us to relate the multi- nature of the problem to associated programming components and solving modules.

Our environment allows domain experts to focus on expressing the models, rather than delving into implementation details, programmers to effectively select the most appropriate available software module for a particular component (subdomain) of the problem w.r.t. its associated single physics model and users to efficiently deploy and run MDMP computations on loosely coupled distributed and heterogeneous compute engines.

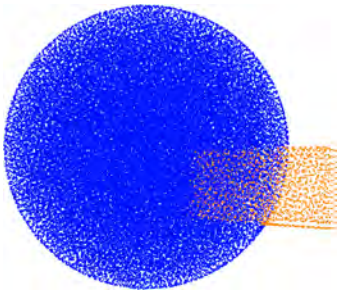
We also show how to exploit remote functionality from machines over a network in a consistent and transparent way to the end user. Our generic design allows us to exploit state of the art software libraries and explore new solving approaches for MDMP problems, with different domain decomposition techniques with or without overlapping.

Appendix A

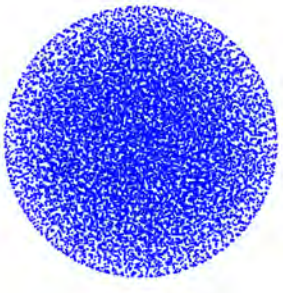
Examples

A.1 3D Schwarz method

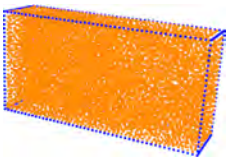
Listing A.3 shows the complete code of a 3D problem. Figure A.1a shows the composite domain, consisting of two subdomains, a sphere and a box, shown in figures A.1b, and A.1c respectively. The definition of the two subdomains is shown in listings A.1 A.2.



(A) The 3D domain consisting of 2 overlapping subdomains



(B) The sphere3D_1 subdomain



(C) The box3D_1 subdomain

FIGURE A.1: Domain decomposition

sphere3D_1.py

```
1  from dolfin import *
2  import solverconfig
3
4  ################################################
5  ###                  config                  ###
6  ################################################
7  # Box's edge points
8  _x = [ -2, 2 ]
9  _y = [ -1, 1 ]
10 _z = [ -.5, .5 ]
11
12 # Sphere's center and radius
13 _c = [ -3, 1, 1 ]
14 _r = 4
15
16 resC = 32
17
18 ################################################
19 ###                  Toolbox                 ###
20 ################################################
21 def Laplacian(expr,x,y,z):
22     dx = diff(expr,x)
23     dx2 = diff(dx,x)
24
25     dy = diff(expr,y)
26     dy2 = diff(dy,y)
27
28     dz = diff(expr,z)
29     dz2 = diff(dz,z)
30
31     dx2dy2dz2 = dx2 + dy2 + dz2
32     return dx2dy2dz2
33
34 ################################################
35 ###         solver API                       ###
36 ################################################
37
38 def ExtBC(x,on_boundary):
39     return on_boundary and not (    between(x[0],(_x[0],_x[1]))
40                                 and between(x[1],(_y[0],_y[1]))
41                                 and between(x[2],(_z[0],_z[1])))
42
43 def ExtIface(x,on_boundary):
44     return on_boundary and (    between(x[0],(_x[0],_x[1]))
45                             and between(x[1],(_y[0],_y[1]))
46                             and between(x[2],(_z[0],_z[1])))
47
```

```
48 class Problem(solverconfig.ConfigCommonProblem):
49     def init(self,*args,**kwargs):
50         mesh_filename = kwargs.get('mesh_filename')
51         mesh = None
52         if not mesh_filename:
53             # the user creates a custom mesh inside this method
54             domain = Sphere(Point(_c[0],_c[1],_c[2]),_r)
55             mesh = Mesh(domain,resC)
56         else:
57             mesh = Mesh(mesh_filename)
58 
59         _ex = [ -4, 4 ]
60         _ey = [ -2, 2 ]
61         _ez = [ -1, 1 ]
62 
63         x = variable(Expression("x[0]"))
64         y = variable(Expression("x[1]"))
65         z = variable(Expression("x[2]"))
66 
67         self.exact = (x-_ex[0])*(x-_ex[1])
68                     *(y-_ey[0])*(y-_ey[1])
69                     *(z-_ez[0])*(z-_ez[1])
70 
71         self.V = FunctionSpace(mesh,'Lagrange',1)
72         u = TrialFunction(self.V)
73         v = TestFunction(self.V)
74 
75         f = -Laplacian(self.exact,x,y,z)
76 
77         self.a = inner(grad(u), grad(v))*dx
78         self.L = f*v*dx
79 
80     def neighbors(self):
81         interface = {}
82         interface['box3D_1'] = ExtIface
83         return interface
84 
85     def boundaries(self):
86         fixed_bc_expr = self.exact
87         bc = DirichletBC(self.V, fixed_bc_expr, ExtBC)
88         return [ bc ]
```

Listing A.1: sphere3D_1.py

box3D_1.py

```
1  from dolfin import *
2  import solverconfig
3
4  ################################################
5  ###                 config                   ###
6  ################################################
7  # Box's edge points
8  _x = [ -2, 2 ]
9  _y = [ -1, 1 ]
10 _z = [ -.5, .5 ]
11
12 # Sphere's center and radius
13 _c = [ -3, 1, 1 ]
14 _r = 4
15
16 resR = 64
17
18 ################################################
19 ###                 Toolbox                  ###
20 ################################################
21 def Laplacian(expr,x,y,z):
22     dx = diff(expr,x)
23     dx2 = diff(dx,x)
24
25     dy = diff(expr,y)
26     dy2 = diff(dy,y)
27
28     dz = diff(expr,z)
29     dz2 = diff(dz,z)
30
31     dx2dy2dz2 = dx2 + dy2 + dz2
32     return dx2dy2dz2
33
34 ################################################
35 ###               solver API                 ###
36 ################################################
37 def ExtBC(x,on_boundary):
38     R = sqrt(  (x[0]-_c[0])*(x[0]-_c[0])
39              + (x[1]-_c[1])*(x[1]-_c[1])
40              + (x[2]-_c[2])*(x[2]-_c[2]))
41     return on_boundary and R >= _r
42
43 def ExtIface(x,on_boundary):
44     R = sqrt(  (x[0]-_c[0])*(x[0]-_c[0])
45              + (x[1]-_c[1])*(x[1]-_c[1])
46              + (x[2]-_c[2])*(x[2]-_c[2]))
47     return on_boundary and R <= _r
```

```
48
49 class Problem(solverconfig.ConfigCommonProblem):
50     def init(self,*args,**kwargs):
51         mesh_filename = kwargs.get('mesh_filename')
52         mesh = None
53         if not mesh_filename:
54             # the user creates a custom mesh inside this method
55             domain = Box(_x[0],_y[0],_z[0],_x[1],_y[1],_z[1])
56             mesh = Mesh(domain,resR)
57         else:
58             mesh = Mesh(mesh_filename)
59
60         _ex = [ -4, 4 ]
61         _ey = [ -2, 2 ]
62         _ez = [ -1, 1 ]
63
64         x = variable(Expression("x[0]"))
65         y = variable(Expression("x[1]"))
66         z = variable(Expression("x[2]"))
67
68         self.exact = (x-_ex[0])*(x-_ex[1])
69                     *(y-_ey[0])*(y-_ey[1])
70                     *(z-_ez[0])*(z-_ez[1])
71
72         self.V = FunctionSpace(mesh,'Lagrange',1)
73         u = TrialFunction(self.V)
74         v = TestFunction(self.V)
75
76         f = -Laplacian(self.exact,x,y,z)
77
78         c = Constant(1.0)
79         self.a = inner(grad(u), grad(v))*dx
80         self.L = f*v*dx + c*self.exact*v*dx
81         #self.L = f*v*dx
82
83     def neighbors(self):
84         interface = {}
85         interface['sphere3D_1'] = ExtIface
86         return interface
87
88     def boundaries(self):
89         fixed_bc_expr = self.exact
90         bc = DirichletBC(self.V, fixed_bc_expr, ExtBC)
91         return [ bc ]
```

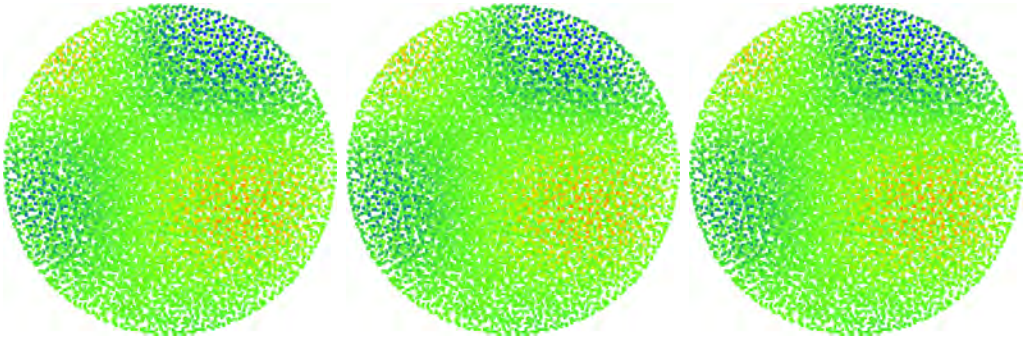
Listing A.2: box3D_1.py

problem3D_1.py

```
1  from dolfin import *
2  import hybridmc as hmc
3
4  import sphere3D_1 as sphere
5  import box3D_1 as box
6
7  use_remote_client = False
8  wsdl_url = 'http://localhost:8000/?wsdl'
9  if use_remote_client:
10     client = hmc.RemoteClient(wsdl_url)
11     client.set_options(timeout=90)      # ****    IMPORTANT    ****
12 else:
13     client = hmc.LocalClient()
14
15 config = hmc.IterativeSolverConfig.Config3D()
16
17 sp = sphere.Problem(priority=1)
18 bp = box.Problem()
19 subdomains=[ sp, bp ]
20
21 sol = client.IterativeSolver(subdomains=subdomains,config=config)
22
23 interactive()
```

Listing A.3: problem3D_1.py

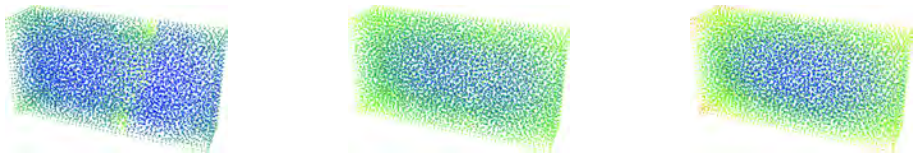
Figure A.3 and A.2 show some plots of the solutions in both subdomains, at the beginning and ending of the iterative solver.



(a) Iteration 1 (b) Iteration 2 (c) Iteration 38 (last)

Figure A.2: Plots from the solution over the sphere subdomain

Figure A.4 shows the convergence rate between the current and previous iteration of the iterative algorithm.



(A) Iteration 1 (B) Iteration 2 (C) Iteration 38 (last)

FIGURE A.3: Plots from the solution over the box subdomain

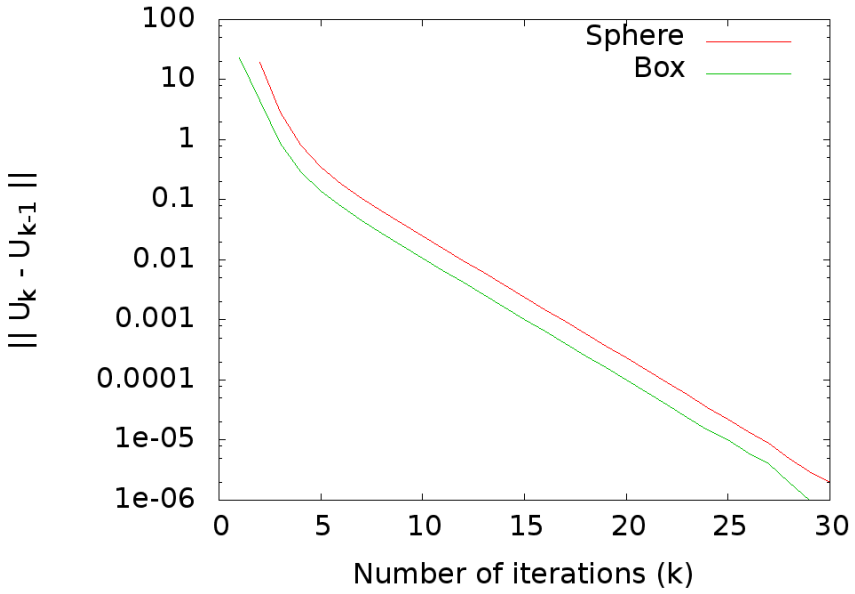


FIGURE A.4: Convergence rate of the two subdomains

A.2 Application setup of chapter 8

Listing A.4 shows the common configuration file for all subdomains. It defines helper functions to express the interfaces between subdomains, as well as any other common input. Listing A.5 shows the setup of the top subdomain of the problem. The rest subdomains have similar definitions. Listing A.6 shows the main file that drives the execution of the Schwarz method of the platform, using as input the defined subdomains.

```
from dolfin import *

""" Definition of Omega domain and helper boundary functions

  A-------------------------------------------B
  |                                           |
  |           subdomain-top    line_Y         |
  |                            |              |
  |                            |              |
  +Z0----------------------ZY+--+Z1----------+Z2-------- line_Z
  |                            |  |           |
  |                            |  |           |
  C--------------------------Y0+--D-----------E
  |                            |  |           |
  |                            |  |           |
  |   subdomain-lmiddle        |  | subdomain |
  |                            |  |   -rmiddle|
  |                            |  |           |
  |                            |  |           |
  F------------G--+X1-------Y1+--H-------------K
  |            |  |           |  |           |
  +W0--------W1+--+WX-------WY+--+W2----------+W3-------- line_W
  |            |  |                          |
  | subdomain  |  |                          |
  |   -lbottom |  |                          |
  |            |  |      subdomain-rbottom   |
  |            |  |                          |
  M------------N--+X0-------------------------P
                  |
                  |
                line_X
"""

# Vertices that describe the subdomain setup
A = [    0. , 3000. ]
B = [ 7000. , 3000. ]
C = [    0. , 1900. ]
D = [ 6000. , 1900. ]
E = [ 7000. , 1900. ]
F = [    0. , 1200. ]
G = [ 2600. , 1200. ]
H = [ 6000. , 1200. ]
K = [ 7000. , 1200. ]
M = [    0. ,    0. ]
N = [ 2600. ,    0. ]
P = [ 7000. ,    0. ]

```

```
48 line_Z = 2300.0
49 line_W =  800.0
50 line_X = 3000.0
51 line_Y = 4600.0
52
53 ZY = [ line_Y, line_Z ]
54 WY = [ line_Y, line_W ]
55 WX = [ line_X, line_W ]
56
57 Z0 = [ C[0] , line_Z ]
58 Z1 = [ D[0] , line_Z ]
59 Z2 = [ E[0] , line_Z ]
60
61 W0 = [ F[0], line_W ]
62 W1 = [ G[0], line_W ]
63 W2 = [ H[0], line_W ]
64 W3 = [ K[0], line_W ]
65
66 X0 = [ line_X, N[1] ]
67 X1 = [ line_X, G[1] ]
68
69 Y0 = [ line_Y, D[1] ]
70 Y1 = [ line_Y, H[1] ]
71
72 def horizontal(l,r,x,on_boundary):
73     return on_boundary and between(x[0],(l[0],r[0])) and near(x[1],l[1])
74 def vertical(b,t,x,on_boundary):
75     return on_boundary and between(x[1],(b[1],t[1])) and near(x[0],b[0])
76
77 # horizontal boundaries
78 def edge_AB(x,on_boundary):
79     return horizontal(A,B,x,on_boundary)
80 def edge_MN(x,on_boundary):
81     return horizontal(M,N,x,on_boundary)
82 def edge_NP(x,on_boundary):
83     return horizontal(N,P,x,on_boundary)
84
85 # horizontal interfaces
86 def edge_CD(x,on_boundary):
87     return horizontal(C,D,x,on_boundary)
88 def edge_DE(x,on_boundary):
89     return horizontal(D,E,x,on_boundary)
90 def edge_GH(x,on_boundary):
91     return horizontal(G,H,x,on_boundary)
92 def edge_FG(x,on_boundary):
93     return horizontal(F,G,x,on_boundary)
94 def edge_HK(x,on_boundary):
95     return horizontal(H,K,x,on_boundary)
```

```
96  def edge_CY0(x,on_boundary):
97      return horizontal(C,Y0,x,on_boundary)
98  def edge_Y0E(x,on_boundary):
99      return horizontal(Y0,E,x,on_boundary)
100 def edge_Z0Z1(x,on_boundary):
101     return horizontal(Z0,Z1,x,on_boundary)
102 def edge_W0WX(x,on_boundary):
103     return horizontal(W0,WX,x,on_boundary)
104 def edge_WXW2(x,on_boundary):
105     return horizontal(WX,W2,x,on_boundary)
106 def edge_ZYZ2(x,on_boundary):
107     return horizontal(ZY,Z2,x,on_boundary)
108 def edge_WYW3(x,on_boundary):
109     return horizontal(WY,W3,x,on_boundary)
110 def edge_FX1(x,on_boundary):
111     return horizontal(F,X1,x,on_boundary)
112 def edge_MX0(x,on_boundary):
113     return horizontal(M,X0,x,on_boundary)
114 def edge_GY1(x,on_boundary):
115     return horizontal(G,Y1,x,on_boundary)
116
117 # vertical boundaries
118 def edge_CA(x,on_boundary):
119     return vertical(C,A,x,on_boundary)
120 def edge_MF(x,on_boundary):
121     return vertical(M,F,x,on_boundary)
122 def edge_EB(x,on_boundary):
123     return vertical(E,B,x,on_boundary)
124 def edge_FC(x,on_boundary):
125     return vertical(F,C,x,on_boundary)
126 def edge_KE(x,on_boundary):
127     return vertical(K,E,x,on_boundary)
128 def edge_PK(x,on_boundary):
129     return vertical(P,K,x,on_boundary)
130
131 # vertical interfaces
132 def edge_HD(x,on_boundary):
133     return vertical(H,D,x,on_boundary)
134 def edge_NG(x,on_boundary):
135     return vertical(N,G,x,on_boundary)
136 def edge_W2Z1(x,on_boundary):
137     return vertical(W2,Z1,x,on_boundary)
138 def edge_W0Z0(x,on_boundary):
139     return vertical(W0,Z0,x,on_boundary)
140 def edge_WYZY(x,on_boundary):
141     return vertical(WY,ZY,x,on_boundary)
142 def edge_W3Z2(x,on_boundary):
143     return vertical(W3,Z2,x,on_boundary)
```

```
144 def edge_X0X1(x,on_boundary):
145     return vertical(X0,X1,x,on_boundary)
146 def edge_Y1K(x,on_boundary):
147     return vertical(Y1,K,x,on_boundary)
148 
149 _K = [ 25.0, 35.0, 50.0, 75.0, 50.0 ]
150 Q_k = [252.0, 450.0, 749.0, 1045.0, 1270.0]
151 mesh_resolution = 20
152 
153 class Delta(Expression):
154     def __init__(self, eps):
155         self.eps = eps
156     def eval(self, values, x):
157         eps = self.eps
158         area=400.0
159         if x[0]==2600.0 and x[1]==1500.0 :
160             values[0] = eps-Q_k[0]/area
161         elif x[0]==3300.0 and x[1]==2200.0 :
162             values[0] = eps-Q_k[1]/area
163         elif x[0]==3900.0 and x[1]==900.0 :
164             values[0] = eps-Q_k[2]/area
165         elif x[0]==4600.0 and x[1]==2400.0 :
166             values[0] = eps-Q_k[3]/area
167         elif x[0]==4800.0 and x[1]==1600.0 :
168             values[0] = eps-Q_k[4]/area
169         else:
170             values[0] = eps
```

Listing A.4: domain_config.py

```
1  from dolfin import *
2  import hybridmc.IterativeSolverConfig as conf
3  import domain_config as Omega
4  
5  class Problem(conf.ConfigCommonProblem):
6      def init(self,*args,**kwargs):
7          # subdomain parameters
8          _ebl = Omega.C
9          _etr = Omega.B
10         _K = Omega._K[0]
11         _eref_bl = Omega.M
12         _eref_tr = Omega.B
13 
14         mesh_filename = kwargs.get('mesh_filename')
15         mesh = None
16         if not mesh_filename:
17             # user-defined mesh
18             nx = int(abs(_etr[0] - _ebl[0])/Omega.mesh_resolution)
```

```
19              ny = int(abs(_etr[1] - _ebl[1])/Omega.mesh_resolution)
20
21              mesh = RectangleMesh(Point(_ebl[0], _ebl[1]),
22                                   Point(_etr[0], _etr[1]),
23                                   nx, ny, "right")
24          else:
25              # load mesh from file
26              mesh = Mesh(mesh_filename)
27
28          self.mesh = mesh
29          parameters["reorder_dofs_serial"] = False
30
31          class Right(SubDomain):
32              def inside(self, x, on_boundary):
33                  return near(x[0], 7000)
34
35          right=Right()
36          boundaries = FacetFunction("size_t", mesh)
37          boundaries.set_all(0)
38          right.mark(boundaries,1)
39          ds=Measure("ds")[boundaries]
40
41          self.V = FunctionSpace(mesh,'Lagrange',1)
42          u = TrialFunction(self.V)
43          v = TestFunction(self.V)
44
45          x = variable(Expression("x[0]"))
46          y = variable(Expression("x[1]"))
47
48          g_r=1.23
49          delta= Omega.Delta(0.03/365.0)
50
51          self.a = inner(_K*grad(u), grad(v))*dx
52          self.L = inner(delta, v)*dx + g_r*v*ds(1)
53
54      def neighbors(self):
55          # create an empty dictionary
56          interface = {}
57          interface['subdomain_lmiddle'] = Omega.edge_CD
58          interface['subdomain_rmiddle'] = Omega.edge_DE
59          return interface
60
61      def boundaries(self):
62          fixed_bc_expr = 0.0
63          bc_left = DirichletBC(self.V, fixed_bc_expr, Omega.edge_CA)
64          return  [ bc_left ]
```

Listing A.5: subdomain_top.py

```
from dolfin import *
import hybridmc as hmc
import sys
import subdomain_top as top
import subdomain_lmiddle as lmiddle
import subdomain_rmiddle as rmiddle
import subdomain_lbottom as lbottom
import subdomain_rbottom as rbottom

######################################################################
######################    create client    ###########################
######################################################################
if len(sys.argv) >= 2:
    port = 8888
    timeout = 90  # ****    IMPORTANT    ****
    if len(sys.argv) >= 3:
        port = int(sys.argv[2])
    if len(sys.argv) >= 4:
        timeout = int(sys.argv[3])
    wsdl_url = "http://%s:%d/?wsdl" %(sys.argv[1],port)
    print wsdl_url
    client = hmc.RemoteClient(wsdl_url)
    client.set_options(timeout=timeout)
else:
    client = hmc.LocalClient()

######################################################################
#################    create subdomain objects    #################
######################################################################
s1 = top.Problem(client=client)
s2 = lmiddle.Problem(client=client)
s3 = rmiddle.Problem(client=client)
s4 = lbottom.Problem(client=client)
s5 = rbottom.Problem(client=client)

######################################################################
###################    create configuration    ###################
######################################################################
config = hmc.IterativeSolverConfig.Config2D(max_iter=15
                                            ,tol_prev=10e-4
                                            ,show_convergence_plots=True
                                            # ,show_interpolant_plots=True
                                            # ,show_solution_plots=True
                                            # ,save_interpolant_plots=True
                                            # ,save_solution_plots=True
                                            # ,log_to_files=True
                                           )
```

```
48 subdomains = [ s1, s2, s3, s4, s5 ]
49
50 ######################################################################
51 ######################    call the solver    ######################
52 ######################################################################
53 solutions = hmc.IterativeSolver(subdomains=subdomains,config=config)
54
55 ######################################################################
56 ###################    plot the solutions    ###################
57 ######################################################################
58 for s, d in zip(solutions,subdomains):
59     plot(s, title=d.__module__)
60
61 interactive()
```

Listing A.6: driver.py

Bibliography

[1] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. ISBN 978-3-642-23098-1. doi: 10.1007/978-3-642-23099-8.

[2] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and T. D. Young. The `deal.ii` library, version 8.1. *arXiv preprint, http://arxiv.org/abs/1312.2266v4*, 2013.

[3] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

[4] Manolis Vavalis. My older papers. 08 2013. URL http://dx.doi.org/10.6084/m9.figshare.774605.

[5] Manolis Vavalis. IMPLEMENTING HYBRID PDE SOLVERS. 08 2014. URL http://dx.doi.org/10.6084/m9.figshare.1134520.

[6] X. Cai. Overlapping domain decomposition methods. In HansPetter Langtangen and Aslak Tveito, editors, *Advanced Topics in Computational Partial Differential Equations*, volume 33 of *Lecture Notes in Computational Science and Engineering*, pages 57–95. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-01438-6. doi: 10.1007/978-3-642-18237-2_2. URL http://dx.doi.org/10.1007/978-3-642-18237-2_2.

[7] Wikipedia. Finite element method, 2014. URL http://en.wikipedia.org/wiki/Finite_element_method. Online; accessed 17-October-2014.

[8] L Traikov, I Antonov, E Dzhambazova, A Ushiama, and Chiyoji Ohkubo. 12 computational fluid dynamics for studying the effects of emf on model systems. *Electromagnetic Fields in Biology and Medicine*, page 173, 2015.

[9] Anders Logg and Garth N. Wells. Dolfin: Automated finite element computing. *ACM Transactions on Mathematical Software*, 37(2), 2010. doi: 10.1145/1731022.1731030.

[10] Martin S. Alnæs. *UFL: a Finite Element Form Language*, chapter 17. Springer, 2012.

[11] Richard A Shapiro. *Adaptive finite element solution algorithm for the Euler equations*, volume 32. Vieweg+ Teubner Verlag, 2013.

[12] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[13] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/1089014.1089021.

[14] Boost basic linear algebra library, 2014. URL http://www.boost.org/doc/libs/release/libs/numeric/ublas/doc/index.htm.

[15] Overview of mtl4, 2014. URL http://www.mtl4.org.

[16] David M. Beazley. Swig: An easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267498.1267513.

[17] M. Vavalis. Implementing hybrid pde solvers. In *Proceedings of the Eleventh International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, 2014.

[18] IEEE. *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System*

Application: Program Interface (API) [C Language]. 1996. ISBN 1-55937-573-6. URL http://standards.ieee.org/reading/ieee/std_public/description/posix/9945-1-1996_desc.html.

[19] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010. ISSN 0740-7475. doi: 10.1109/MCSE.2010.69. URL http://dx.doi.org/10.1109/MCSE.2010.69.

[20] Wikipedia. Laplace operator — wikipedia, the free encyclopedia, 2014. URL http://en.wikipedia.org/w/index.php?title=Laplace_operator&oldid=610975119. [Online; accessed 21-October-2014].

[21] Paul F. Dubois, Konrad Hinsen, and James Hugunin. Numerical python. *Computers in Physics*, 10(3), May/June 1996.

[22] Thomas Williams, Colin Kelley, and many others. Gnuplot 4.6: an interactive plotting program, 2014. URL http://gnuplot.sourceforge.net.

[23] Wikipedia. Soap — wikipedia, the free encyclopedia, 2014. URL http://en.wikipedia.org/w/index.php?title=SOAP&oldid=630165480. [Online; accessed 23-October-2014].

[24] Wikipedia. Web services description language — wikipedia, the free encyclopedia, 2014. URL http://en.wikipedia.org/w/index.php?title=Web_Services_Description_Language&oldid=614428231. [Online; accessed 23-October-2014].

[25] Arskom Ltd. spyne - rpc that doesn't break your back., 2014. URL http://spyne.io. [Online; accessed 23-October-2014].

[] Wikipedia. Model–view–controller — wikipedia, the free encyclopedia, 2016. URL https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&oldid=698792507. [Online; accessed 15-January-2016].

[26] Fedorahosted.org. Suds is a lightweight soap python client for consuming web services., 2014. URL https://fedorahosted.org/suds. [Online; accessed 23-October-2014].

[27] P. Stratis, Y. Saridakis, M. Zakynthinaki, and E. Papadopoulou. Alopex stochastic optimization for pumping management in fresh water coastal aquifers. In *Journal of Physics: Conference Series*, volume 490. 2014.

[28] P. Stratis, G. Karatzas, E. Papadopoulou, M. Zakynthinaki, and Y. Saridakis. Stochastic optimization for an analytical method of saltwater intrusion on coastal aquifers. submitted to PLOSone, 2015.

[29] P. Stratis, Z. Dokou, G. Karatzas, E. Papadopoulou, and Y. Saridakis. Stochastic optimization and numerical simulation for pumping management of the hersonissos freshwater coastal aquifer in crete. In *Procs of 2015 Int. Conf. on Water Resources, Hydraulics and Hydrology, Zakynthos, Greece*, pages 329–334.

[30] A. Mantoglou, M. Papantoniou, and P. Giannoulopoulos. Management of coastal aquifers based on nonlinear optimization and evolutionary algorithms. *Journal of Hydrology*, 297:209–228, 2004.

[31] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, May-Jun 2007.